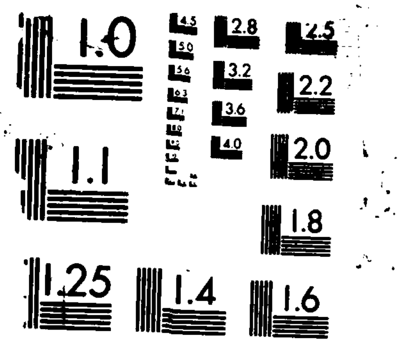
AD-A188 151   SEMANTICS OF EQL(U) NORTH CSROLINA UNIV AT CHAPEL HILL   1/1
DEPT OF COMPUTER SCIENCE B JAYARAMAN 1987
N00014-86-K-0680

UNCLASSIFIED   F/G 12/5   NL



MICROCOPY RESOLUTION TEST CHART

Semantics of EqL†

Bharat Jayaraman
Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC 27514

Abstract—We presents the formal semantics of a novel language, called EqL, for first-order functional and Horn logic programming. An EqL program is a set of conditional pattern-directed rules, where the conditions are expressed as a conjunction of equations. The programming paradigm provided by this language may be called *equational programming*. The declarative semantics of equations is given in terms of their *complete set of solutions*, and the operational semantics for solving equations is an extension of *reduction*, called *object refinement*. The correctness of the operational semantics is established through *soundness* and *completeness* theorems. Examples are given to illustrate the language and its semantics.

Index Terms—Functional programming, Logic programming, Equational programming, Denotational semantics, Reduction semantics, Object refinement, Equation solving, Correctness theorems.

† This research is supported by grant DCR-8603609 from the National Science Foundation and contract N 00014-86-K-0680 from the Office of Naval Research.

I. INTRODUCTION

The integration of functional and logic languages has received considerable interest recently, and a number of different approaches have been proposed since Robinson's LOGLISP in the late seventies [18]. We propose a new approach, offering a simple and uniform framework for first-order functional and Horn logic programming. Our approach is in contrast to some recent approaches [7, 5, 14] which permit both functions and predicate clauses, thereby resulting in two different programming styles within one language. The major advantage of having a uniform framework is the simplicity in the semantics of the resulting language. This framework is an extension of functional languages in two respects:

1. The *pattern-directed rule* is augmented with a *set of equations*, for specifying constraints on variables.

2. The *reduction* model of execution is augmented with a technique called *object refinement*, for solving equations.

We choose pattern-directed rules, because they form the basis of many modern functional languages [8, 12, 15, 22]. By extending these rules with conditions expressed as a conjunction of equations, we have shown that the capability of Horn-logic can be achieved [9, 10]. In the language EqL (for Equational Language) described in this paper, a program consists of a set of conditional pattern-directed rules, followed by a top-level goal which is a set of equations to be solved. EqL supports functional programming because a functional expression e to be evaluated is treated as an equation $v = e$, where v is some distinct variable. EqL also supports logic programming because a goal g to be solved can also be viewed as an equation, namely, $g = true$. Because solutions are not necessarily unique, the declarative semantics of a set of equations is expressed in terms of its *complete set of solutions* relative to a given program. In our semantics, constructors and function symbols are sharply distinguished, and the denotation of a ground expression is a set of terms over the constructors—a *set* is needed because of the possibility of nondeterminism.

It was our objective to define the operational semantics of the language in terms of *reduction*, because of its suitability for parallel execution. However, solving for variables in equations requires an extension of reduction, which we call *object refinement*. An equation is thus solved by progressively reducing its two expressions and "refining" the data objects bound to the variables. This process may be seen as a restriction of *narrowing* [6, 5, 3, 20, 24] that provides an efficient search for solutions. It also gives many opportunities for parallel execution, including ***and-parallelism*** and ***or-parallelism*** [1].

The correctness of the operational semantics is established through ***soundness*** and ***completeness*** theorems. Essentially, the soundness theorem says that any solution computed according to the operational semantics is as general as a solution defined by the declarative semantics, whereas the completeness theorem says that any solution defined by the declarative semantics has a solution computed by the operational semantics that is as general.

The rest of this paper divides into the following sections: section II introduces the language EqL, and presents examples of programs for functional and Horn-logic programming; sections III and IV define respectively the declarative and operational semantics of EqL programs; correctness issues are addressed in section V; finally, section VI presents conclusions and further comparisons with related work.

II. EqL: AN EQUATIONAL LANGUAGE

A. Language Features

In the syntactic definition below, the symbols ***opname***, ***atom***, and ***variable*** stand for user-defined lexical units.

program ⟶ *rules goal*

rules ⟶ ϵ | *rule rules*

rule ⟶ *opname* (*terms*) => *body* | *opname* () => *body*

goal ⟶ *equations*

terms ⟶ *term* | *term* , *terms*

term ⟶ *atom* | *variable* | cons(*term* , *term*)

body ⟶ *expression* | *expression* where *equations*

expression ⟶ *atom* | *variable* | cons(*expression* , *expression*)
| *opname*(*actuals*) | *opname*()

actuals ⟶ *expression* | *expression* , *actuals*

equations ⟶ *equation* | *equation* , *equations*

equation ⟶ *expression* = *expression*

We use the word ***term*** and ***pattern*** synonymously. A ***term*** refers to a data object built up from the constructor cons, atoms, and variables. A ***ground term*** is a term without any variables. Our convention is that all atoms are quoted, excepting numerals. As in LISP, cons builds a binary tree. Lists are a special case of trees, but we write them in EqL using the [...] notation, similar to Prolog. Although we use a single constructor, cons, our approach is applicable to other constructors as well. The element [ ] is considered an atom, and stands for the empty tree and also the empty list.

Unlike ordinary functional languages, the rules in EqL need not be deterministic. Another difference is that the right-hand side of an EqL rule may contain variables not present on its left-hand side. All variables in a rule are *logical variables*, similar to Prolog, in the sense that their bindings must satisfy certain constraints. (Note that there are no global variables in this language.) A set of equations forms a conjunctive set, i.e., they must be simultaneously ***satisfiable***. We define satisfiability formally in the next section; informally, an equation $e_1 = e_2$ is considered satisfiable if there is a substitution, γ, of ground terms for the variables in e_1 and e_2, such that the sets of ground terms denoted by $(e_1\ \gamma)$ and $(e_2\ \gamma)$ have a common term. Finally, it should be noted that all operations, as well as the constructor cons, are ***strict***, i.e., their result is undefined, or $\perp$, if any of their arguments is undefined.

B. Examples

We now present a few simple examples to show the capabilities of these constructs for functional and logic programming.

Example 1: Append

Below is the EqL definition for the familiar LISP function append.

```
append([ ], y) => y
append([h | t], y) => [h | append(t, y)]
?  answer = append([1,2], [3,4])
```

The top-level goal is an equation for evaluating the expression `append([1,2], [3,4])`. The output is the value of the variable `answer`, namely, `[1,2,3,4]`.

Example 2: Pure Prolog in EqL

A simple example is given to illustrate the mechanical conversion of any pure Prolog, or Horn logic, program to an EqL program. The following program is written using DEC-10 Prolog syntax [2]:

```
rev([ ], [ ]).
rev([H|T], Y) :- rev(T, Z), app(Z, [H], Y).
? rev(X, [1,2,3,4]).
```

The mechanically derived EqL program would be:

```
rev([ ], [ ]) => true
rev([H | T], Y) => true where rev(T, Z) = true
                              app(Z, [H], Y) = true
? rev(X, [1,2,3,4]) = true
```

The basic idea is to treat each Prolog predicate as a true-valued EqL operation. In general, a predicate p defined by k clauses would be translated into k rules. The set of equations in each rule serves to solve the goals in the clause body. Note that the variable Z on the right-side of the second rule for rev does not appear on the corresponding left-side. This feature of EqL in fact is crucial for attaining the expressiveness of Horn logic.

It should be noted that the above program is not the best way to define the 'reverse' operation in EqL. For example, the program below is a clearer definition of 'reverse':

```
reverse([ ]) => [ ]
reverse([h | t]) => append(reverse(t), [h])
? reverse(x) = [1,2,3,4]
```

The operation append is as defined in Example 1.

Example 3: Permutations

To conclude this section, we present a nondeterministic program for producing the permutations of a list:

```
perm([ ]) => [ ]
perm(x) => [e | perm(delete(e, x))]
delete(h, [h | t]) => t
delete(h1, [h2 | t]) => [h2 | delete(h1, t)]
? answer = perm([1,2,3,4])
```

Note, again, that the variable e on the right-side of the second rule for perm does not appear on the left-side of the rule. The variable answer has 24 possible bindings, corresponding to the 24 permutations of the list [1,2,3,4].

III. DECLARATIVE SEMANTICS

In this section, we describe a denotational semantics [21] for EqL programs. The syntactic domains are abbreviated as follows: A for Atom, Var for Variable, Exp for Expression, and Eqn for Equation. The semantic domains are: G for Ground Term, Env for Environment, and T for the domain with a single value, *true*. Each *syntactic* ground term has a corresponding element in G. We use italics to distinguish the semantic element from the syntactic element, e.g., *cons* is the semantic equivalent of cons. An environment is a mapping of variables to ground terms, Env = [Var $\rightarrow$ G]. In addition, we introduce the domain $T_\perp = \{\perp, true\}$, with $\perp \sqsubseteq true$.

The semantics of expressions and equations are expressed by the following two semantic functions:

μ_P: Exp $\rightarrow$ Env $\rightarrow$ G $\rightarrow$ $T_\perp$

ν_P: Eqn $\rightarrow$ Env $\rightarrow$ $T_\perp$

Our approach is to define μ_P in such a way that $\mu_P[\![e]\!]\sigma g$ yields *true* if ground term g is a possible value for expression e in environment σ (relative to an EqL program P). Similarly, the semantic function ν_P is defined so that $\nu_P[\![S]\!]\sigma$ yields *true* if the set of equations $S = [\![d_1 = e_1, \ldots, d_n = e_n]\!]$ is *satisfiable* in environment σ (relative to an EqL program P). In the next two subsections, we define the semantic functions μ_P and ν_P respectively.

1. Expressions

The semantic function μ_P is defined for each of the four kinds of expressions: atoms, variables, cons-expressions, and operation applications. In the definition below, we assume a semantic function K which maps a *syntactic* atom a to its semantic counterpart.

$$\mu_P[\![a]\!]\sigma g = \begin{cases} true & \text{if } K(a) = g; \\ \perp & \text{otherwise.} \end{cases}$$

$$\mu_P[\![x]\!]\sigma g = \begin{cases} true & \text{if } \sigma x = g; \\ \perp & \text{otherwise.} \end{cases}$$

$$\mu_P[\![\text{cons}(e_1, e_2)]\!]\sigma g = \begin{cases} \mu_P[\![e_1]\!]\sigma g_1 \ \wedge \ \mu_P[\![e_2]\!]\sigma g_2 & \text{if } g = \text{cons}(g_1, g_2); \\ \perp & \text{otherwise.} \end{cases}$$

$$\mu_P[\![f(e_1, \ldots, e_n)]\!]\sigma g = \sqcup_i \ \sqcup_{\gamma \in \text{Env}} \ (\sigma \sqsubseteq \gamma \ \wedge \ \mu_P[\![exp_i]\!]\gamma g \ \wedge \ \nu_P[\![S_i]\!]\gamma \ \wedge \ \nu_P[\![T_i]\!]\gamma),$$

where

$f(t_{i1}, \ldots, t_{in})$ => exp_i where S_i is the i-th rule for f, and

T_i is the set of equations $[\![t_{i1} = e_1, \ldots, t_{in} = e_n]\!]$.

In the last case above, we say that g is a possible ground value for expression $f(e_1, \ldots, e_n)$ in environment σ if there is some rule for f, say

$f(t_{i1}, \ldots, t_{in})$ `=>` exp_i **where** S_i,

such that g is a possible value for $[\![exp_i]\!]$ in a more defined environment γ, provided that the set of equations $[\![S_i]\!]$ and $[\![T_i]\!]$ are *satisfiable* in environment γ (defined in the next subsection). The environment γ is an extension of σ in that it additionally binds all variables introduced by the i-th rule for f to ground terms. We assume that all variables in the i-th rule for f are renamed to avoid naming conflicts. We define $\sigma \sqsubseteq \gamma = (\forall x)\sigma x \sqsubseteq \gamma x$. (Note that γ may be expressed as $\sigma \cup \eta$, where η binds only the variables in f.) We define the connective $\wedge$ as follows: $true \wedge true = true$, and $x \wedge \perp = \perp \wedge x = \perp$.

2. Equations

The semantic function ν_P defined below expresses what it means for a set of equations, $S = [\![d_1 = e_1, \ldots, d_n = e_n]\!]$, to be *satisfiable* in an environment σ.

$$\nu_P[\![S]\!]\sigma = (\forall i) \ \sqcup_{g_i \in G} \ (\mu_P[\![d_i]\!]\sigma g_i \ \wedge \ \mu_P[\![e_i]\!]\sigma g_i).$$

Basically, an equation $d_i = e_i$ is satisfiable in environment σ if there a common ground term, g_i, that is a possible value for both d_i and e_i. A set of equations is considered satisfiable if each equation in the set is satisfiable; otherwise the set of equations is said to be *unsatisfiable*.

Because the top-level goal equations may contain variables, we define the *solution* to a set equations as a substitution of ground terms for the variables of the equations that makes the equations satisfiable. For a set of equations S involving only *terms*, there is a unique *most general* solution (involving non-ground terms possibly), which is the *most general unifier* of S. For a set of equations S that also involves operation applications, e.g., `append(x, y) = [1,2,3,4]`, there need not be any unique most general solution. We therefore introduce the notion of the *complete set of solutions* of a set of equations.

Definition 1: The *complete set of solutions* of a set of equations S with respect to a program P is $\Sigma_P = \{\sigma \mid \nu_P[\![S]\!]\sigma\}$, where σ binds all variables in S to ground terms.

3. Operation Symbols

We can now define the meaning of an operation symbol f by means of the semantic function $\mathcal{G}_P$, as follows:

$$(\mathcal{G}_P[\![f]\!])(\mathrm{G}_1,\ldots,\mathrm{G}_n) = \cup_{g_1 \in \mathrm{G}_1} \cdots \cup_{g_n \in \mathrm{G}_n} \{g \mid \mu_P[\![f(g_1,\ldots,g_n)]\!]\bot_{\mathrm{Env}} g\}$$

where each $\mathrm{G}_i \subset \mathrm{G}$ and $\mathrm{G}_i \neq \phi$, the empty set. We use $\bot_{\mathrm{Env}}$ to denote the undefined environment, i.e., one which does not specify bindings for any variables. If some $\mathrm{G}_i = \phi$, then we define $(\mathcal{G}_P[\![f]\!])(\mathrm{G}_1,\ldots,\mathrm{G}_n) = \phi$, which specifies that all operations are *strict*. The ordering for the domain of each operation is $\phi \sqsubseteq g$, for each $g \subset \mathrm{G}$.

Suppose we define the meaning of a *ground* expression e with the semantic function $\mathcal{G}_P$ as follows:

$$\mathcal{G}_P[\![e]\!] = \{g \mid \mu_P[\![e]\!]\bot_{\mathrm{Env}} g\}.$$

We then have the following *substitution theorem*:

Theorem 1:

$$\mathcal{G}_P[\![f(e_1,\ldots,e_n)]\!] = (\mathcal{G}_P[\![f]\!])(\mathcal{G}_P[\![e_1]\!],\ldots,\mathcal{G}_P[\![e_n]\!]),$$

where each e_i is a ground expression.

4. Discussion

In the interest of brevity, we omit a detailed proof of theorem 1. It follows from simplifying the two sides of the equality, and observing that the meaning of *ground* expression e_i in the undefined environment, $\bot_{\mathrm{Env}}$, is the same as in any other environment γ. Although the definitions of μ_P and ν_P are recursive, because they are expressed as a composition of continuous functions, we can take their meaning to be the least fixed point of the recursive transformation. Because the definition of $\mathcal{G}_P$ is *not* recursive, it is possible to use sets and unions without requiring a complex power-domain construction [17].

An important point about our semantics is that the meaning of an equation $e_1 = e_2$ is based on the *denoted values* of the two expressions e_1 and e_2. To illustrate its implication, consider the two operations defined below:

```
f(x) => cons(x,10)
loop(x) => loop(x)
```

Given these rules, the goal equation

```
? f(5) = f(z)
```

does have a solution, namely, z ← 5. However, the equation

```
? loop(10) = loop(y)
```

does not have any solution according to our semantics, because the expression `loop(10)` does not denote any value. If, as in *equational logic* [24], the semantics of expressions is *not* based on the denoted values, the above equation does have a solution, namely, y ← 10, which is obtained by *syntactically unifying* the two expressions. This example reveals one of the key differences between our *domain-theoretic* semantics and *equational logic* semantics.

Finally, we should point out that the semantics defined by $\mathcal{G}_P$ does not distinguish between an operation such as `f`, defined above, which always terminates, and an operation such as g, defined below, which fails to terminate:

```
g(x) => cons(x, 10)
g(x) => loop(x)
```

Distinguishing between `f` and g semantically would require extending the input and output sets of each operation with $\perp$, which would stand for nontermination. An output set {`cons(1,10)`} would then be different from the set {$\perp$,`cons(1,10)`}.

IV. OPERATIONAL SEMANTICS

In this section, we first develop a *reduction semantics*, which we show to be equivalent to the declarative semantics of the previous section. We then develop a *refinement semantics* which can serve as an effective operational strategy for an interpreter. This refinement semantics also forms the operational semantics of EqL.

A. Reduction

1. Notation and Terminology

In order to express the solution of equations using reduction rules, we represent an equation $d_1 = e_1$ as a single expression $\mathcal{E}(d_1, e_1)$, where $\mathcal{E}$ is a unique constructor. In general, a set of equations $[\![d_1 = e_1, \ldots, d_n = e_n]\!]$, is represented by a binary tree of nested *and*'s as follows:

$$and(\mathcal{E}(d_1, e_1), \ldots, and(\mathcal{E}(d_{n-1}, e_{n-1}), \mathcal{E}(d_n, e_n))).$$

In order to refer to parts of an equation, we define, similar to Hullot [6], an *occurrence* as a string consisting of a l's and r's (for left and right subtree respectively), which uniquely identifies a subterm of the "equation tree." For example, the third equation, $\mathcal{E}(d_3, e_3)$, in the above set of equations is at occurrence rrl. Similarly, the expression e_3 is at occurrence rrlr. Since we are not interested in occurrences within an expression, it suffices for us to use l and r rather than natural numbers.

Given a set of equations S, the notation $S[o]$ refers to an equation eqn_1 at occurrence o. The notation $S[o \leftarrow eqn_2]$ refers to the set of equations S with eqn_1 at occurrence o replaced by eqn_2.

2. The $\rightarrow$ relation

We express the reduction semantics of an EqL program in terms of the reduction relation $\rightarrow$. This is a one-to-many relation over equations. We use the value *true* from domain T to denote the satisfaction of an equation.

i. *Unit Identity:*

$$\mathcal{E}(a,a) \rightarrow true \text{ IF } atom(a)$$

ii. *And Elimination:*

$$and(true, true) \rightarrow true$$

iii. *Decomposition:*

$$\mathcal{E}(cons(e_1,e_2), cons(e_3,e_4)) \rightarrow and(\mathcal{E}(e_1,e_3), \mathcal{E}(e_2,e_4))$$

iv. *Operation Application With Variable Binding:*

$$\mathcal{E}(f(e_1,\ldots,e_n),e) \rightarrow and(\mathcal{E}(exp\ \eta, e), and(S\eta, \mathcal{T}\eta))$$

$$\mathcal{E}(e, f(e_1,\ldots,e_n)) \rightarrow and(\mathcal{E}(exp\ \eta, e), and(S\eta, \mathcal{T}\eta))$$

if there is a rule $f(t_1,\ldots,t_n)$ `=>` exp `where` S, and

$\mathcal{T}$ is the set of equations, $t_1 = e_1, \ldots, t_n = e_n$, and

η binds all variables in the rule for f to ground terms.

A rule of the form $f(t_1,\ldots,t_n)$ `=>` exp is just syntactic sugar for $f(t_1,\ldots,t_n)$ `=>` exp `where` *true*.

Note that rule i does not reduce two identical expressions to *true*, except in the case of identical atoms. Two identical ground *terms* can be detected to be identical using rules i, ii, and iii. We do not provide a primitive identity check on two arbitrary but identical *operation applications*, e.g. `loop(10) = loop(10)`, because the applications might not denote any value.

Given a set of equations S, if an equation eqn_1 at occurence o rewrites to eqn_2, i.e., $eqn_1 \rightarrow eqn_2$, we write

$$S \rightarrow S[o \leftarrow eqn_2].$$

The reader may note that we have used the same notation $\rightarrow$ to denote two relations: one between two equations, and the other between two sets of equations. However, from context it will be easy to distinguish which of the two relations is intended.

A derivation starting from S and using rules i-iv is called a *$\rightarrow$-derivation*, and is denoted by $\rightarrow^*$.

Theorem 2: The ***complete set of solutions*** of a set of equations S with respect to a program P, $\Sigma_P = \{\sigma \mid S\sigma \rightarrow^* true\}$, where σ binds all variables in S to ground terms.

Proof Sketch: To prove the equality of the two sets, we need to show that each set is included in the other. The proofs for the two cases are similar, hence we illustrate only one of them. For this proof, we assume, without loss of generality, that all equations are expressed in the form $v = exp$, where v is a variable. Note that an equation $exp_1 = exp_2$ can always be rewritten using the two equations: $v = exp_1, v = exp_2$, where v is a distinct variable. The main case of interest is an equation involving an operation application, which we can assume, without loss of generality, to be of the form $v = f(e_1, \ldots, e_n)$.

Suppose that $S\sigma \rightarrow^* true$. We wish to show $\nu_P[\![S]\!]\sigma$. Consider an equation $v = f(e_1, \ldots, e_n)$ in S, and let the first reduction step involve the k-th rule for f and some substitution η for the variables in f. In the declarative semantics, we may choose σv for the value of g_i in the application of ν_P, and we may choose the same k-th rule for f and $\gamma = \sigma \cup \eta$ in the application of μ_P. Now, the resulting expressions in the $\rightarrow$-reduction and in the declarative semantics are equivalent. In this manner we can create equivalent expressions at any reduction step by an application of rule iv. Because $S\sigma$ eventually $\rightarrow$-reduces to *true*, it follows that $\nu_P[\![S]\!]\sigma$ can also be simplified to *true*.

B. Refinement

Rules i-iii can be used directly in an interpreter, but rule iv cannot, because suitable values for η cannot be determined during operation application. Therefore, in defining the operational semantics, we use rule iv with $\eta = \phi$, the empty substitution, and we introduce a set of rules for computing bindings for variables. We shall refer to rule iv used in this way as rule v, *Operation Application Without Variable Binding*.

1. The $\rightsquigarrow$ Relation

The computation of bindings for variables is defined by a new relation, denoted $\rightsquigarrow$. Rule vi below defines the $\rightsquigarrow$ relation.

vi. *Variable Binding:*

There are basically two sets of cases: (a) the binding of a variable or an atom to a variable and (b) the binding of a cons-term to a variable. The rules for case (a) are:

$\mathcal{E}(v, a) \leadsto_\sigma$ *true* where $\sigma = \{v \leftarrow a\}$

$\mathcal{E}(a, v) \leadsto_\sigma$ *true* where $\sigma = \{v \leftarrow a\}$

$\mathcal{E}(v_1, v_2) \leadsto_\sigma$ *true* where $\sigma = \{v_1 \leftarrow z, v_2 \leftarrow z\}$

$\mathcal{E}(v, v) \leadsto_\phi$ *true* where ϕ is the empty substitution

where v, v_1 and v_2 are variables (v_1 and v_2 are distinct), a is an atom, and z is some distinct new variable. The rules for case (b) are:

$\mathcal{E}(v, \mathrm{cons}(e_1, e_2)) \leadsto_\sigma and(\mathcal{E}(z_1, e_1), \mathcal{E}(z_2, e_2))$

$\mathcal{E}(\mathrm{cons}(e_1, e_2), v) \leadsto_\sigma and(\mathcal{E}(z_1, e_1), \mathcal{E}(z_2, e_2))$

where v is a variable, $\sigma = \{v \leftarrow cons(z_1, z_2)\}$, and z_1 and z_2 are two distinct variables.

Note that we do not permit the binding of an arbitrary expression to a variable for the same reason we do not permit an identity check on two arbitrary expressions. Given rules i-iii, v and vi, we are ready to define the reduction rules for a *set of equations* S. We associate *solution bindings* σ with a set of equations S, and denote this pair by (S with σ). These bindings associate a term with each variable in the equations. Initially these bindings are empty, thus initially we have (S with ϕ). As the set of equations get reduced, the associated solution bindings get "refined." The following two rules express this refinement:

1. (S with ρ_1) $\leadsto$ ($S[o \leftarrow eqn]$ with ρ_1) IF $S[o] \rightarrow eqn$

2. (S with ρ_1) $\leadsto$ ($(S[o \leftarrow eqn])\rho_2$ with $\rho_1 \cup \rho_2$) IF $S[o] \leadsto_{\rho_2} eqn$

Note that ρ_1 and ρ_2 are disjoint. A derivation starting from (S with ϕ) and using rules i-iii, v, and rule vi is called a $\leadsto$*-derivation*, and is denoted by $\leadsto^*$.

Definition 2: $\sigma \downarrow V(S)$ is a ***computed solution*** of a set of equations S if there is a $\leadsto$-derivation of (*true* with σ) starting from (S with ϕ), i.e.,

(S with ϕ) $\leadsto^*$ (*true* with σ),

where $\sigma \downarrow V(S)$ refers to the subsitution σ restricted to the variables in S.

Note that we have altogether introduced four relations: the relation $\rightarrow$ between two equations, the relation $\rightarrow$ between two *sets* of equations, the relation $\leadsto_\rho$ between two equations, and the the relation $\rightsquigarrow$ between two *sets* of equations. The relation $\rightarrow$ between two equations and the relation $\rightsquigarrow_\rho$ are mutually exclusive.

2. Example

We illustrate the operational semantics by a simple example. Consider the following program:

```
    null([ ]) => true

    null(cons(x1, x2)) => false
```

and a top-level equation

```
    ?  null(y) = true
```

The top-level equation is expressed using the constructor $\mathcal{E}$ as $\mathcal{E}$(null(y), true). A $\rightsquigarrow$-derivation of *true* for this equation is shown below:

$\mathcal{E}$(null(y), true) with ϕ

$\rightsquigarrow$ *and*($\mathcal{E}$(y, [ ]), $\mathcal{E}$(true, true)) with ϕ (by rule v)

$\rightsquigarrow$ *and*($\mathcal{E}$(y, [ ]), *true*) with ϕ (by rule i)

$\rightsquigarrow$ *and*(*true*, *true*) with $\{y \leftarrow [\,]\}$ (by rule vi)

$\rightsquigarrow$ *true* with $\{y \leftarrow [\,]\}$ (by rule ii)

The above example uses the first definition of null in the application of rule v. Another $\rightsquigarrow$-derivation is possible using the second rule for null, but this derivation will not terminate with *true*. Note that true and false are syntactic elements, whereas *true* $\in$ T is a semantic element. Also, $[\,]$ is the semantic equivalent of [ ].

3. Discussion

The solution of a set of equations is thus a process of gradually "refining" the values of its variables until *true* is derived. This process is therefore referred to as *object refinement.* It is a generalization of the reduction rule in functional languages because refinement includes reduction. The $\rightsquigarrow$ relation can serve as the basis for a breadth-first procedure for finding the complete set of solutions. At each step of the rewriting process, any equation may selected for rewriting. If an operation application appears at the outermost level in one of its constituent expressions it would be rewritten to all of its different possible right-hand sides, thereby resulting in multiple sets of equations to be solved for multiple solutions. This gives rise to *or*-parallelism, in the terminology logic programming languages [1]. Note that multiple solutions can arise only because of the presence of multiple definitions for some operation.

In order to prune unproductive computational paths, it is necessary to recognize the following forms of equations in an equation set, where the e_i's are arbitrary expressions.

$\mathcal{E}(a_1, a_2)$ IF $atom(a_1) \wedge atom(a_2)$
$\mathcal{E}(a, \text{cons}(e_1, e_2))$ IF $atom(a)$
$\mathcal{E}(\text{cons}(e_1, e_2), a)$ IF $atom(a)$
$\mathcal{E}(e, f(e_1, \ldots e_n))$ IF f has no definition
$\mathcal{E}(f(e_1, \ldots e_n), e)$ IF f has no definition

An equation-set containing any of the above forms of equations can never reduce to *true*, and therefore may be abandoned. Indeed, an interpreter should be on the look out for these cases and cause early pruning of such branches. Deferring operation application until no other kinds of rules apply can also avoid much over-computation.

V. CORRECTNESS

We now address the correctness of the operational semantics defined in the previous section via soundness and completeness theorems. The soundness theorem basically states that every computed solution is as general as some member of the complete set of solutions, whereas the completeness theorem states that every solution belonging to the complete set of solutions has a computed solution that is as general. We discuss the implication of these results at end of this section.

A. Soundness

Theorem 3 (Soundness): Given a set of equations S and a program P, if there is a $\rightsquigarrow$-derivation

$$S \text{ with } \phi \rightsquigarrow^* true \text{ with } \rho$$

then there exists a $\sigma \in \Sigma_P$, where $\rho \sqsubseteq \sigma$.

Proof Sketch: Because of the equivalence between Σ_P and $\{\sigma \mid S\sigma \rightarrow^* true\}$, it suffices to show that there exists a $\rightarrow$-derivation $S\sigma \rightarrow^* true$. We prove this theorem by *construction*; that is, we derive a $\rightarrow$-derivation from the given $\rightsquigarrow$-derivation. The basic approach is as follows: We seek to replace all $\rightsquigarrow$ steps in the $\rightsquigarrow$-derivation by $\rightarrow$ steps. That is, we seek to replace all type vi steps (Variable binding) by type i (Unit identity) and type iii (Decomposition) steps while maintaining a correct derivation. In the process, some type v steps (Operation application without variable binding) may become type iv steps (Operation application with variable binding), although the bindings may not be fully ground yet. The only form of type vi step that cannot be replaced is a step $\mathcal{E}(v,v) \rightsquigarrow_{\phi}$ *true*. Thus, when all other type vi steps have been replaced, a derivation $S\rho \rightarrow^{*} \mathcal{T}$ will result, where $\mathcal{T}$ consists only of equations of the form $\mathcal{E}(v,v)$, where v is a variable either in the goal S or introduced by some rule. By replacing each variable v with an arbitrary atom, we obtain a derivation $S\sigma \rightarrow^{*}$ *true*, where $\rho \sqsubseteq \sigma$. Further details of the replacements are given in [9].

B. Completeness

Before presenting the completeness theorem, we introduce two definitions: For two expressions e_1 and e_2, we define $e_1 \sqsubseteq e_2$ if there is a substitution ρ binding variables to terms such that $e_1\rho = e_2$. In a similar manner, for two sets of equations S and $\mathcal{T}$, $S \sqsubseteq \mathcal{T}$ if there is a ρ such that $S\rho = \mathcal{T}$.

Theorem 4 (Completeness): Given a set of equations S and a program $\mathcal{P}$, and a solution $\sigma \in \Sigma_{\mathcal{P}}$, or equivalently, $S\sigma \rightarrow^{*}$ *true*, there exists ρ such that

$$S \text{ with } \phi \rightsquigarrow^{*} \textit{true} \text{ with } \rho$$

where $\rho \downarrow V(S) \sqsubseteq \sigma$.

Proof Sketch: From the given $\rightarrow$-derivation, we construct a $\rightsquigarrow$-derivation in which each equation-set is at least as general as the corresponding equation-set in the $\rightarrow$-derivation; and in so doing we will build up the solution bindings $\rho \sqsubseteq \sigma$. The crux of the proof involves showing that the $\rightsquigarrow$-derivation can find solutions without rule iv, Operation application with variable binding. We show this by induction on the number of steps in the $\rightarrow$-derivation.

We first abbreviate $S\sigma$ by $\mathcal{T}$. Let $\mathcal{T}_i$ be the i-th equation-set in the given $\rightarrow$-derivation and S_i be i-th equation-set in $\rightsquigarrow$-derivation to be constructed. For the base case i=1, $S_i \sqsubseteq \mathcal{T}_i$ (because $S \sqsubseteq S\sigma$), and the solution bindings $\phi \sqsubseteq \sigma$. For the induction hypothesis, we assume that for all $k \leq i$, $S_k \sqsubseteq \mathcal{T}_k$ and the partial solution $\rho_k \sqsubseteq \sigma$. We now extend the partial $\rightsquigarrow$-derivation one step and show that we can rewrite S_i to S_{i+1} such that $S_{i+1} \sqsubseteq \mathcal{T}_{i+1}$ and $\rho_{i+1} \sqsubseteq \sigma$. Showing this entails a case analysis of the four rules (i-iv) used to derive $\mathcal{T}_{i+1}$ from $\mathcal{T}_i$. The details are provided in [9]; the main points are: (i) if rule iv is used in the $\rightarrow$-derivation, the corresponding step in the $\rightsquigarrow$-derivation will use rule v; (ii) if rules i (Unit identity) or iii (Decomposition) are used in the $\rightarrow$-derivation, the corresponding step in the $\rightsquigarrow$-derivation could require rule vi (Variable binding). In each case, we show that $S_{i+1} \sqsubseteq T_{i+1}$ and $\rho_{i+1} \sqsubseteq \sigma$, using the induction hypothesis. Because the $\rightarrow$-derivation ends in *true*, so also must the constructed $\rightsquigarrow$-derivation. And because the bindings produced by the application of rule vi may involve nonground terms, the solution ρ computed in the $\rightsquigarrow$-derivation may be more general than σ.

C. Discussion

From the standpoint of an interpreter of EqL, the soundness theorem guarantees that no incorrect solutions will be produced if the interpreter followed the rules defined in the operational semantics. Completeness guarantees that all solutions can be found, but it does not guarantee that one can construct a solution procedure that will always terminate, reporting that all solutions have been found. Because of the possibility of nonterminating definitions, in general the solution of arbitrary equations will not necessarily terminate. We can be sure that the interpreter has produced all solutions only if it terminates. Our completeness result thus corresponds to *recursively enumerable completeness* [5]. It is possible to devise sufficient conditions for proving termination, in the manner of Hullot, Plaisted and others [6, 3], but we have not yet investigated this issue. Finally, it should be understood that the computed solutions are most general in the sense that any other solution can be expressed as an instance of one of the computed solutions.

VI. CONCLUSIONS AND RELATED WORK

Although EqL supports functional programming more directly than logic programming because of its functional syntax, we hope it is clear that logic programming paradigms are also readily stated in the language. In fact, any Horn-logic program can be directly converted into an EqL program. Because equation solution lies at the heart of the language, the programming paradigm provided by the EqL may be called *equational programming.*

Hoffman and O'Donnell first introduced the term *equational programming* [8, 16] to refer to a style of programming with equations whose semantics are based on the logical consequences of equality. The extension of O'Donnell's language for logic programming by You and Subrahmanyam [24] permits a set of equations to be solved only at the top-level. The rules in EqL are more general in that they permit a set of equations to appear on the right-hand side of a rule as well. Dershowitz and Plaisted have recently used the term equational programming to refer to a style of programming with conditional rewrite rules [3] that provides the capability of first-order functional and Horn-logic programming in a uniform and elegant way. The language of oriented equational clauses proposed by Fribourg [4] is also related. An important difference in our approach from other equational languages is that the formal semantics of our language is *domain-theoretic*, rather than based on *equational logic* [4, 16, 24].

A number of different approaches have been proposed for the operational semantics of a language combining functional and logic capabilities [3, 4, 5, 13, 20]. Of these, the closest related approach is *narrowing* in term-rewriting systems [3, 5, 6, 20, 24]. Two variations of narrowing have recently been proposed: *conditional narrowing* [3] and *lazy narrowing* [20]. Object refinement combines the generality of conditional narrowing (in that it is applicable to conditional rules) with the efficiency of lazy narrowing (in that reductions occur at the outermost level of an expression). Because of EqL's strict semantics, the data objects computed by object refinement are always finite; the "laziness" in object refinement is with respect to failure. Object refinement also bears a close resemblance to the equation solution procedure in SUPER [19]: its 'decomposition' and 'instantiation' rules are very similar to our decomposition and variable binding rules. The main difference is that SUPER is based on lambda calculus whereas our approach is based equation rewriting.

In order that we may concentrate on semantics, we have restricted the language we have presented to its essential constructs: terms over the single constructor cons, expressions, equations, and operation definitions. In practice, it is necessary to enhance the language by permitting user-definable constructors, primitives for arithmetic, and domain testing predicates such as LISP's *atom*, *numberp*, etc. These extra primitives, however, would be used only for ordinary reduction, but not for object refinement. Examples of programs using these extensions were given in [11]. We have implemented this extended language in order to demonstrate the feasibility of these ideas.

ACKNOWLEDGMENTS

My sincere thanks to the following people: David Plaisted, for many suggestions in the development of the language constructs and semantics of EqL; Frank Silbermann, for assistance with the operational semantics and correctness proofs; Dean Brock, for helpful discussions on denotational semantics; and, one of the anonymous referees for constructive criticism and detailed suggestions on improving the denotational semantics of section III. The comments and suggestions of the other referees were also helpful in improving this presentation.

REFERENCES

[1] J. S. Conery and D. F. Kibler, "Parallel Interpretation of Logic Programs," In *Conf. Functional Prog. Lang. and Comp. Arch.*, Portsmouth, NH, 1981, pp. 163–170.

[2] W. F. Clocksin and C. S. Mellish, *Programming in Prolog,* Springer-Verlag, New York, 1981.

[3] N. Dershowitz and D. A. Plaisted, "Applicative Programming *cum* Logic Programming," In *1985 Symp. on Logic Programming,* Boston, MA, July 1985, pp. 54–66.

[4] L. Fribourg, "Oriented Equational Clauses as a Programming Language." *J. Logic Prog.*, Vol. 2, pp. 165-177, 1984.

[5] J. A. Goguen and J. Meseguer, "Equality, Types, Modules, and (Why Not?) Generics for Logic Programming," *J. Logic Prog.*, Vol. 2, pp. 179–210, 1984.

[6] J-M. Hullot, "Canonical Forms and Unification," In *Proc. 5th Workshop on Automated Deduction,* Lecture Notes in Computer Science, Springer-Verlag, 1980, pp. 318–334.

[7] A. Hansson, S. Haridi, and S.-A. Tärnlund, "Properties of a Logic Programming Language," In *Logic Programming,* K. L. Clark and S.-A. Tärnlund (Eds.), Academic Press, 1982, pp. 267–280.

[8] C. M. Hoffman and M. J. O'Donnell, "Programming with Equations," *ACM Trans. on Prog. Langs. and Systems*, Vol. 4, No. 1, pp. 83–112, January 1982.

[9] B. Jayaraman and F. S. K. Silbermann, "Equations, Sets, and Reduction Semantics for Functional and Logic Programming," In *1986 ACM Conf. on LISP and Functional Programming,* Boston, MA, Aug. 1986, pp. 320-331.

[10] B. Jayaraman, F.S.K. Silbermann, G. Gupta, "Equational Programming: A Unifying Approach to Functional and Logic Programming," In *Int'l Conf. on Computer Languages*, pp. 47-57, Miami Beach, October 1986.

[11] B. Jayaraman, and G. Gupta, "EqL User's Guide," TR 87-010, Department of Computer Science, Univ. of North Carolina, Chapel Hill, May 1987.

[12] R. M. Keller, "FEL Programmer's Guide," AMPS Tech. Memo. 7, Department of Computer Science, Univ. of Utah, Salt Lake City, April 1982.

[13] G. Lindstrom, "Functional Programming and the Logical Variable," In *12th ACM*

Symp. on Princ. of Prog. Langs., New Orleans, LA, Jan. 1985, pp. 266–280.

[14] Y. Malachi, Z. Manna, and R. Waldinger, "TABLOG: The Deductive-Tableau Programming Language," In *ACM Symp. on LISP and Functional Programming*, Austin, TX, Aug. 1984, pp. 323–330.

[15] R. Milner, "A Proposal for Standard ML," In *ACM Symp. on LISP and Functional Programming*, Austin, TX, Aug. 1984, pp. 184-197.

[16] M. J. O'Donnell, "Equational logic as a programming language," M.I.T. Press, Cambridge, MA, 1985.

[17] G. Plotkin, "A Powerdomain Construction," *SIAM Journal of Computing*, Vol. 5, pp. 452-486, 1976.

[18] J. A. Robinson and E. E. Sibert, "LOGLISP: Motivation, Design, and Implementation," In *Logic Programming*, K. L. Clark and S.-A. Tärnlund (Eds.), Academic Press, 1982, pp. 299–313.

[19] J. A. Robinson, "New Generation Knowledge Processing: Syracuse University Parallel Expression Reduction," First Annual Progress Report, Syracuse University, December 1984.

[20] U. S. Reddy, "Narrowing as the Operational Semantics of Functional Languages," In *1985 Symp. on Logic Programming*, Boston, MA, July 1985, pp. 138–151.

[21] J. E. Stoy, "Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory," MIT Press, Cambridge, Mass., 1977.

[22] D. A. Turner, "Miranda: A non-strict functional language with polymorphic types," in *Conf. on Functional Prog. Langs. and Comp. Arch.*, Nancy, France, Sep. 1985, pp. 1-16.

[23] D. H. D. Warren, F. Pereira, and L. M. Pereira, "Prolog: the Language and Its Implementation Compared with LISP," *SIGPLAN Notices*, Vol 12., No. 8, pp. 109–115, 1977.

[24] J-H. You and P. A. Subrahmanyam, "Equational Logic Programming: an Extension to Equational Programming," In *13th ACM Symp. on Princ. of Prog. Langs.*, St. Petersburg, FL, 1986, pp. 209-218.

END

FEb.

1988

DTIC